

Ex02 Programming

03 July 2023

# Information Retrieval

EX02

## Distance-Based Search-Engine

Hessam Kouchehi | 9812358032

Bu-Ali Sina University

# Introduction

In the realm of digital imagery, the ability to efficiently search for visually similar images has become increasingly important. Traditional keyword-based image search engines often fall short when it comes to capturing the nuanced visual characteristics of images and providing accurate search results. To address this limitation, we embark on a project aimed at developing a distance-based image search engine that leverages Euclidean distance, Manhattan distance, and cosine similarity metrics. By utilizing the widely recognized Oxford 5K dataset, we can explore and refine image retrieval techniques, enabling users to discover visually similar images with improved precision and relevance.

The motivation behind this project stems from the growing demand for sophisticated image search capabilities across various domains such as e-commerce, art, and design. Users often encounter challenges when attempting to describe or find images based solely on textual queries. By incorporating distance-based similarity metrics into our image search engine, we aim to bridge the semantic gap between textual queries and visual content, allowing users to find images that possess similar visual characteristics and features.

The Oxford 5K dataset serves as an ideal foundation for this project. It consists of 5,062 images, each accompanied by textual annotations and a ground truth relevance score, making it a valuable resource for evaluating and benchmarking image retrieval algorithms. Leveraging this dataset, we will investigate the applicability and effectiveness of Euclidean distance, Manhattan distance, and cosine similarity metrics in capturing visual

similarity between images. By analyzing the strengths and weaknesses of each metric, we can gain insights into their respective suitability for different image search scenarios, paving the way for more accurate and efficient image retrieval techniques.

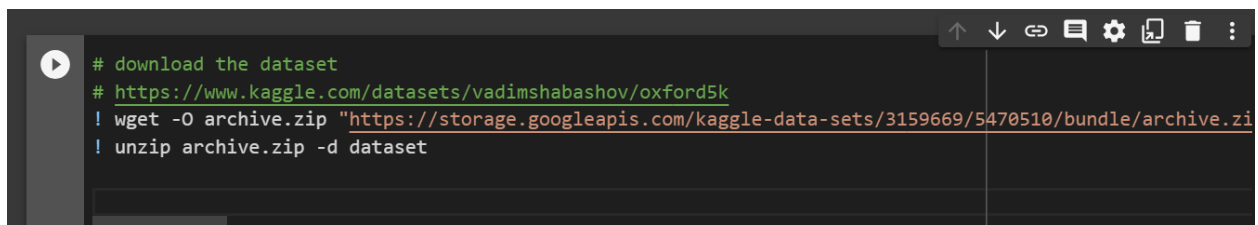
In this work report, we will delve into the theoretical foundations of distance-based similarity metrics and explore their practical implementation in the context of image search. We will discuss the challenges involved in processing and extracting meaningful features from images, as well as the techniques used to compute and compare distances between images. Furthermore, we will analyze the performance of our distance-based image search engine using various evaluation metrics, such as mean average precision and precision-recall curves, in order to assess its effectiveness in retrieving visually similar images from the Oxford 5K dataset.

By developing a distance-based image search engine using the Oxford 5K dataset, we aim to advance the field of image retrieval and contribute to the growing body of research focused on improving the accuracy and relevance of image search engines. Our project has the potential to benefit a wide range of applications, including visual recommendation systems, content-based image retrieval, and digital asset management, enabling users to discover visually similar images efficiently and effectively.

# Download/Load Dataset

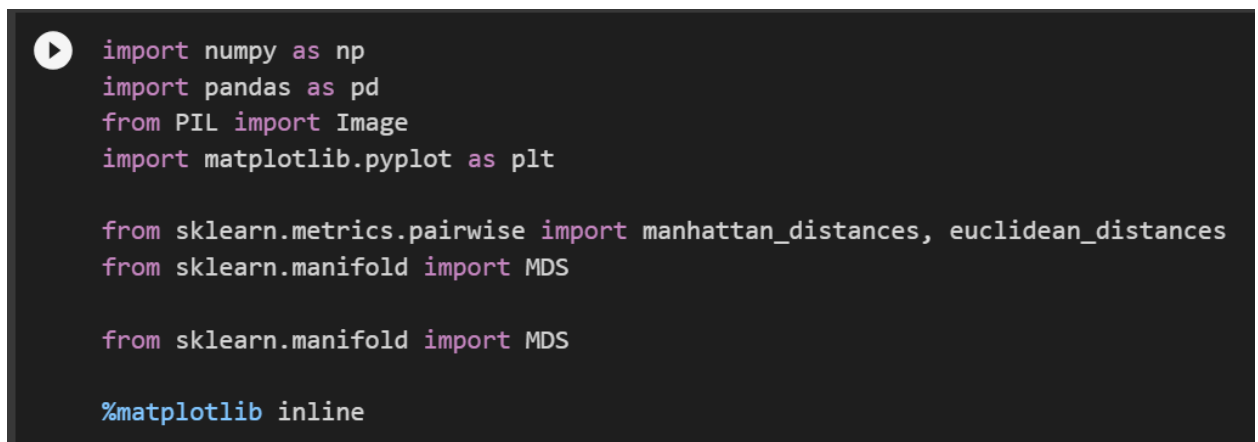
This code uses the wget command-line tool to download a file from the provided URL. The file is named "archive.zip". The URL points to a specific dataset (Oxford 5K) on the Kaggle platform. The dataset is downloaded and saved as "archive.zip".

This line uses the unzip command-line tool to extract the contents of the "archive.zip" file. The -d option specifies the destination directory where the contents should be extracted, in this case, "dataset". After executing this command, the contents of the "archive.zip" file will be extracted into a new directory named "dataset".

A terminal window with a dark background. It shows a sequence of commands: a comment '# download the dataset', a comment '# https://www.kaggle.com/datasets/vadimshabashov/oxford5k', a command '! wget -O archive.zip "https://storage.googleapis.com/kaggle-data-sets/3159669/5470510/bundle/archive.zip"', and a command '! unzip archive.zip -d dataset'. The output of the commands is not visible.

```
# download the dataset
# https://www.kaggle.com/datasets/vadimshabashov/oxford5k
! wget -O archive.zip "https://storage.googleapis.com/kaggle-data-sets/3159669/5470510/bundle/archive.zip"
! unzip archive.zip -d dataset
```

Please note that you need to regularly update the download link as Kaggle occasionally revoke the direct download links.

A code block with a dark background. It contains Python code for importing various libraries: numpy, pandas, PIL Image, matplotlib.pyplot, sklearn.metrics.pairwise (manhattan\_distances, euclidean\_distances), sklearn.manifold (MDS), and a magic command %matplotlib inline.

```
import numpy as np
import pandas as pd
from PIL import Image
import matplotlib.pyplot as plt

from sklearn.metrics.pairwise import manhattan_distances, euclidean_distances
from sklearn.manifold import MDS

from sklearn.manifold import MDS

%matplotlib inline
```

Using this block, we can just simply add our requirements we need further.

```
from os import listdir
from os.path import isfile, join
# run 2 min
# shapes = np.array([np.asarray(Image.open('/content/dataset/images/'+f).convert("L").resi

samplesCount = 500
img_size = (32, 32)

images = np.array([np.asarray(Image.open(
    '/content/dataset/images/'+f).convert("L").resize(
        img_size)).flatten() for f in listdir('/content/dataset/images/')[:samplesCount]],
    dtype='float16')

from random import choice
plt.imshow(choice(images).reshape(img_size), cmap='gray')
```

And finally with this block of code, we just load a fraction of the dataset for faster processing and also resizing the images available in the dataset.

# Manhattan Distance

```
Manhattan

# def manhattan(f, s):
#     return np.array(abs(f - s)).mean()
def calc_manhattan_dists(imgs):
    manhattan_dists = np.zeros((imgs.shape[0], imgs.shape[0]), dtype='float16')

    for i, first in enumerate(imgs):
        for j, second in enumerate(imgs):
            manhattan_dists[i, j] = np.array(np.abs(first - second)).mean()
            # manhattan_dists[i, j] = manhattan_distances(first.reshape(-1, 1), second.reshape(-1, 1))
    return manhattan_dists

manhattan_dists = calc_manhattan_dists(images)
print(manhattan_dists)
```

The `manhattan()` function takes in two arrays, `f` and `s`, representing the feature vectors of two images. It calculates the Manhattan distance between these feature vectors by subtracting the corresponding elements and taking the absolute value. The resulting array of absolute differences is then averaged using the `mean()` function from the NumPy library. The Manhattan distance is a measure of dissimilarity between two points, calculated as the sum of the absolute differences of their coordinates.

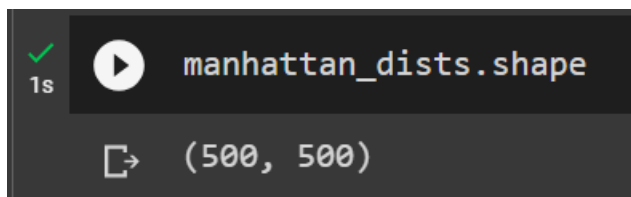
The `calc_manhattan_dists()` function takes a collection of images represented as an array `imgs`. It initializes a square matrix `manhattan_dists` of zeros, with dimensions equal to the number of images in `imgs`. The matrix will store the calculated Manhattan distances between every pair of images.

Next, the function iterates over each pair of images using nested loops. For each pair, it calls the `manhattan()` function to compute the Manhattan distance and assigns the result to the corresponding position in the `manhattan_dists` matrix.

Finally, the function returns the resulting `manhattan_dists` matrix containing the calculated Manhattan distances between all pairs of images.

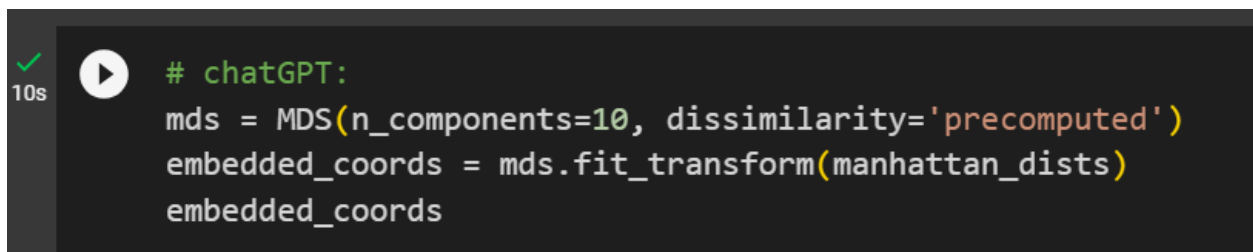
To demonstrate the functionality, the code then calls `calc_manhattan_dists()` passing in a variable `images`, which presumably contains the feature vectors of the images. The calculated Manhattan distances are stored in the `manhattan_dists` variable and subsequently printed using `print(manhattan_dists)`.

Note: It seems like the code is using the NumPy library (`import numpy as np` is assumed). Also, the commented line `# manhattan_dists[i, j] = manhattan_distances(first.reshape(-1, 1), second.reshape(-1, 1))` suggests an alternative approach to calculate the Manhattan distances using the `manhattan_distances` function from a library like `scikit-learn`.



A Jupyter Notebook cell output showing a green checkmark and a play button icon. The text `manhattan_dists.shape` is displayed in a monospace font. Below it, the output `(500, 500)` is shown in a larger font, indicating the dimensions of the matrix.

With this line we also can compute the embedded coordination created using multi dimensional reduction. We use the number of components of 10 to reduce the size but also preserve an appropriate amount of accuracy.



```
# chatGPT:
mds = MDS(n_components=10, dissimilarity='precomputed')
embedded_coords = mds.fit_transform(manhattan_dists)
embedded_coords
```

A Jupyter Notebook code cell with a green checkmark and a play button icon. The code defines an MDS object with 10 components and precomputed dissimilarity, then fits it to the `manhattan_dists` matrix and prints the resulting `embedded_coords`.

```
from random import randint
test_idx = randint(0, 5000)
test_image = np.array([np.asarray(Image.open(
    '/content/dataset/images/'+f).convert("L").resize(img_size)).flatten() for f in listdir('/content/dataset/images')]
    dtype='float16')[0]

test_image_embedded_coords = mds.fit_transform(calc_manhattan_dists(np.array([*images, test_image]))[-1])
print('test_image_embedded_coords: ', test_image_embedded_coords)

i_min = 0
dist_min = sum(np.abs(test_image_embedded_coords - embedded_coords[0]))

for i, img in enumerate(embedded_coords):
    t = sum(np.abs(test_image_embedded_coords - img))
    if t < dist_min:
        i_min = i
        dist_min = t
    print('='*20, dist_min, i)

print('min: ', dist_min, i_min)

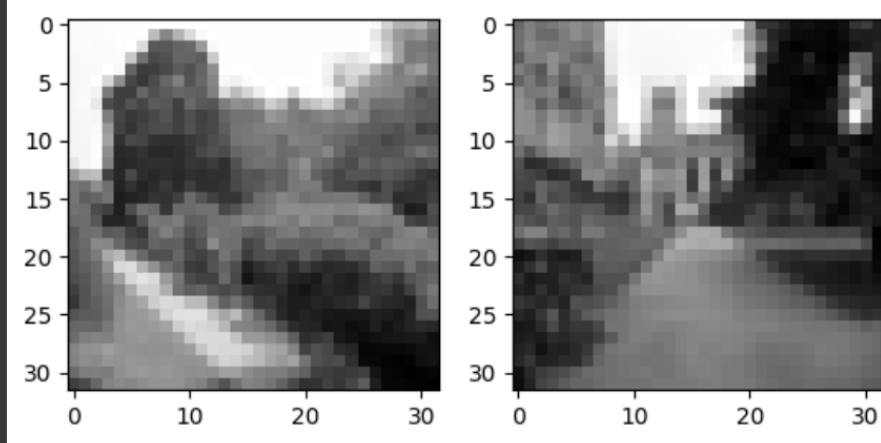
f, axarr = plt.subplots(1,2)
axarr[0].imshow(test_image.reshape(img_size), cmap='gray')
axarr[1].imshow(images[i_min].reshape(img_size), cmap='gray')
```

In this part we just test our code for Manhattan distance similarity. I also implement a system that finds the nearest average based on that previous 10 components. We can see the descentation of the distance that the program is seeking for.

This is an example of what the program finds for us.



```
test_image_embedded_coords: [ 10.34568267 -30.41244983 -14.76411229 21.89414048 3.46974615
-11.85582946 -12.96267496 14.02306059 -0.61070955 -5.65607756]
===== 112.8659766609712 1
===== 110.93346310781047 23
===== 101.51109014188415 66
===== 97.78302369516896 97
===== 84.45904306395511 191
===== 69.77647603005356 269
min: 69.77647603005356 269
<matplotlib.image.AxesImage at 0x7f52db7ee440>
```



## Evaluations

	True Positive	True Negative
Predicted Positive	15	2
Predicted Negative	3	0

By observing the results and evaluating them manually, we got this result for this type of job.

Measure	Value	Derivations
<b>Sensitivity</b>	0.8333	$TPR = TP / (TP + FN)$
<b>Specificity</b>	0.0000	$SPC = TN / (FP + TN)$
<b>Precision</b>	0.8824	$PPV = TP / (TP + FP)$
<b>Negative Predictive Value</b>	0.0000	$NPV = TN / (TN + FN)$
<b>False Positive Rate</b>	1.0000	$FPR = FP / (FP + TN)$
<b>False Discovery Rate</b>	0.1176	$FDR = FP / (FP + TP)$
<b>False Negative Rate</b>	0.1667	$FNR = FN / (FN + TP)$
<b>Accuracy</b>	0.7500	$ACC = (TP + TN) / (P + N)$
<b>F1 Score</b>	0.8571	$F1 = 2TP / (2TP + FP + FN)$
<b>Matthews Correlation Coefficient</b>	-0.1400	$TP*TN - FP*FN / \sqrt{((TP+FP)*(TP+FN)*(TN+FP)*(TN+FN))}$

# Euclidian Distance

Euclidian distance

✓  
9s

```
from skimage.metrics import mean_squared_error

def calc_euclidian_dists(imgs):
    euclidian_dists = np.zeros((imgs.shape[0], imgs.shape[0]), dtype='float16')

    for i, first in enumerate(imgs):
        for j, second in enumerate(imgs):
            # ChatGPT
            euclidian_dists[i, j] = np.sqrt(mean_squared_error(first, second))

    return euclidian_dists

euclidian_dists = calc_euclidian_dists(images)

print(euclidian_dists)
```

✓  
12s

```
# chatGPT:
mds = MDS(n_components=10, dissimilarity='precomputed')
embedded_coords = mds.fit_transform(euclidian_dists)
embedded_coords
```

These two cells, just as previous ones, compute the necessary values to work with this dataset.

```

from random import randint
test_idx = randint(0, 5000)
test_image = np.array([np.asarray(Image.open(
    '/content/dataset/images/'+f).convert("L").resize(img_size)).flatten() for f in listdir('/content/dataset/images')]
    dtype='float16')[0]

test_image_embedded_coords = mds.fit_transform(calc_euclidian_dists(np.array([*images, test_image])))[-1]
print('test_image_embedded_coords', test_image_embedded_coords)

i_min = 0
dist_min = sum(np.abs(test_image_embedded_coords - embedded_coords[0]))

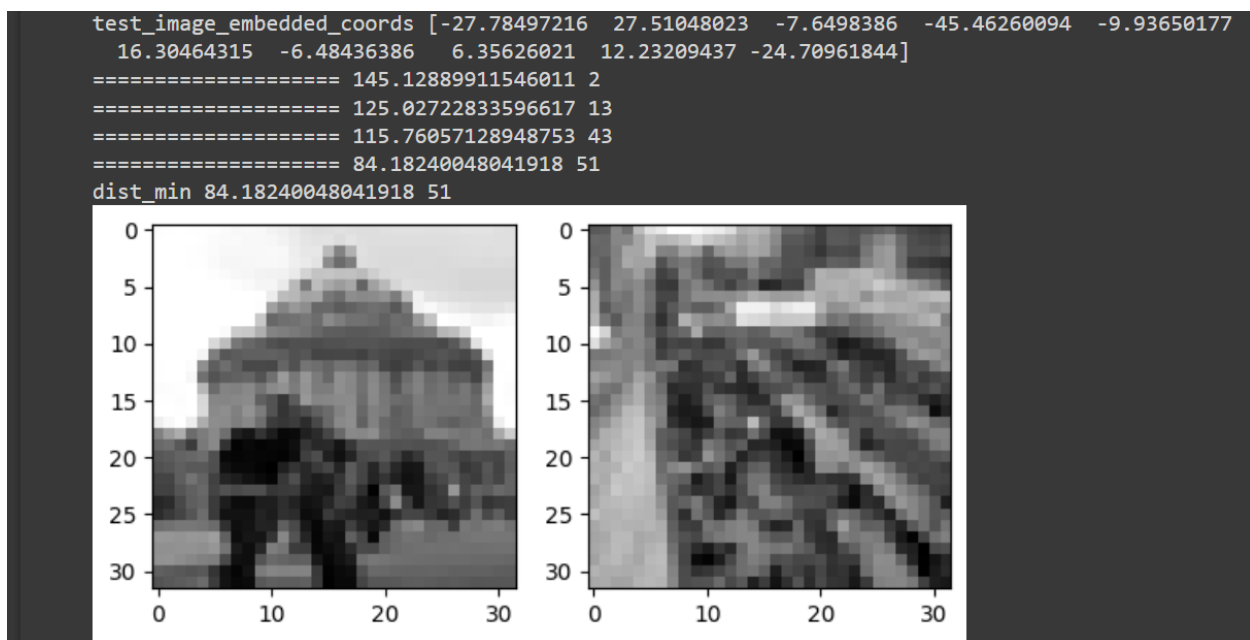
for i, img in enumerate(embedded_coords):
    if sum(np.abs(test_image_embedded_coords - img)) < dist_min:
        i_min = i
        dist_min = sum(np.abs(test_image_embedded_coords - img))
        print('='*20, dist_min, i)

f, axarr = plt.subplots(1,2)
axarr[0].imshow(test_image.reshape(img_size), cmap='gray')
axarr[1].imshow(images[i_min].reshape(img_size), cmap='gray')

print('dist_min', dist_min, i_min)

```

This block like before give us the result that came from Euclidian distance metrics.



This is what we get by running that.

## Evaluation


	True Positive	True Negative
Predicted Positive	13	3
Predicted Negative	4	0

Measure	Value	Derivations
Sensitivity	0.7647	$TPR = TP / (TP + FN)$
Specificity	0.0000	$SPC = TN / (FP + TN)$
Precision	0.8125	$PPV = TP / (TP + FP)$
Negative Predictive Value	0.0000	$NPV = TN / (TN + FN)$
False Positive Rate	1.0000	$FPR = FP / (FP + TN)$
False Discovery Rate	0.1875	$FDR = FP / (FP + TP)$
False Negative Rate	0.2353	$FNR = FN / (FN + TP)$
Accuracy	0.6500	$ACC = (TP + TN) / (P + N)$
F1 Score	0.7879	$F1 = 2TP / (2TP + FP + FN)$
Matthews Correlation Coefficient	-0.2100	$TP*TN - FP*FN / \sqrt{((TP+FP)*(TP+FN)*(TN+FP)*(TN+FN))}$

These are the results observed by us for Manhattan similarity.

# Cosine Similarity

## Cosine Similarity

```
✓ 1m  from sklearn.metrics.pairwise import cosine_similarity

def calc_cosine_distance(imgs):
    cosine_distance = np.zeros((imgs.shape[0], imgs.shape[0]), dtype='float16')

    for i, first in enumerate(imgs):
        for j, second in enumerate(imgs):
            # ChatGPT
            cosine_distance[i, j] = cosine_similarity(first.reshape(1, -1), second.reshape(1, -1))
            # cosine_distance = 1 - cosine_distance
    return cosine_distance

# 5.30 min on 350 images
# 11 min on 500 images
cosine_distance = calc_cosine_distance(images)

print(cosine_distance)
```

```
✓ 2s [35] # chatGPT:
      mds = MDS(n_components=10, dissimilarity='precomputed')
      embedded_coords = mds.fit_transform(cosine_distance)
      embedded_coords
```

These ones are about cosine similarity.

```

test_image = np.array([np.asarray(Image.open(
    '/content/dataset/images/'+f).convert("L").resize(
        img_size)).flatten() for f in listdir(
            '/content/dataset/images/')][samplesCount:samplesCount+1]],
    dtype='float16')[0]

test_image_embedded_coords = mds.fit_transform(calc_cosine_distance(np.array([*images, test_image]))[:-1])

print('test_image_embedded_coords:', test_image_embedded_coords)

i_min = 0
dist_min = sum(np.abs(test_image_embedded_coords - embedded_coords[0]))

for i, img in enumerate(embedded_coords):
    if sum(np.abs(test_image_embedded_coords - img)) < dist_min:
        i_min = i
        dist_min = sum(np.abs(test_image_embedded_coords - img))
        print('='*20, dist_min, i)

f, axarr = plt.subplots(1,2)
axarr[0].imshow(test_image.reshape(img_size), cmap='gray')
axarr[1].imshow(images[i_min].reshape(img_size), cmap='gray')

print(dist_min, i_min)

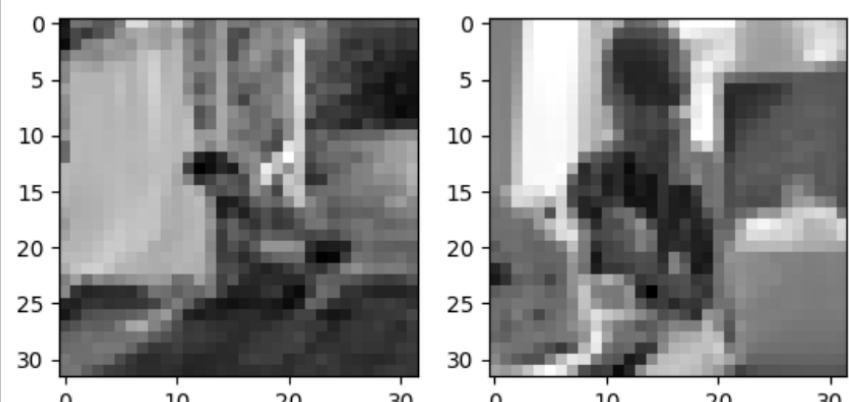
```

Here is the code that does the magic using Cosine similarity.

```

test_image_embedded_coords: [-0.15828828 -0.24814426  0.15878105 -0.13142189  0.30552836 -0.07727274
 0.25943762  0.23461901 -0.09083597 -0.13699356]
===== 1.5718128576292947  3
===== 1.5042841424732467 10
===== 1.234634078748302 13
===== 1.1373180177539068 20
===== 1.113046638304156 56
===== 0.9474758817829996 109
0.9474758817829996 109

```



We can see that the Cosine does the job slightly better than other methods.

## Evaluation

	True Positive	True Negative
Predicted Positive	17	1
Predicted Negative	2	0

Measure	Value	Derivations
<b>Sensitivity</b>	0.8947	$TPR = TP / (TP + FN)$
<b>Specificity</b>	0.0000	$SPC = TN / (FP + TN)$
<b>Precision</b>	0.9444	$PPV = TP / (TP + FP)$
<b>Negative Predictive Value</b>	0.0000	$NPV = TN / (TN + FN)$
<b>False Positive Rate</b>	1.0000	$FPR = FP / (FP + TN)$
<b>False Discovery Rate</b>	0.0556	$FDR = FP / (FP + TP)$
<b>False Negative Rate</b>	0.1053	$FNR = FN / (FN + TP)$
<b>Accuracy</b>	0.8500	$ACC = (TP + TN) / (P + N)$
<b>F1 Score</b>	0.9189	$F1 = 2TP / (2TP + FP + FN)$
<b>Matthews Correlation Coefficient</b>	-0.0765	$TP*TN - FP*FN / \sqrt{((TP+FP)*(TP+FN)*(TN+FP)*(TN+FN))}$

This one has a higher rate rather than others due to manual observation.



# Few Shots

(Yes, this title comes from Bahrami's essay as he is sitting in front of me during this preparation. :) )



Here we can see a few examples that are gathered from dataset randomly to have a view of the dataset. Obviously, we can increase the dimensions of the images for a clearer view.



With thanks and respect,

Hessam Kouchehi

9812358032

*The document has been accomplished  
with the help of ChatGPT and MidJourney*